Using MMX™ Instructions to Transpose a Matrix

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Using MMX™ Instructions to Transpose a Matrix

March 1996

CONTENTS

1.0. INTRODUCTION

Transposing a matrix requires reordering a large amount of data. This application note shows how to use the MMX™ Technology PUNPCK instructions to significantly speed up a matrix transpose operation. The PUNPCK instructions can interleave four 16-bit numbers from two sources - and can do so once per clock. When one of the two sources is an MMX register filled with zeroes, the PUNPCK instructions unpack data from 16-bits to 32-bit numbers (or from 8-bits to 16-bit numbers, or from 32-bits to 64-bit numbers).

In this application note, two methods are presented for transposing a matrix. The primary method replaces the source matrix values with the transposed result. This method produces the highest performance for small matrices (those are fully contained in the data cache of the Pentium® processor). This method is limited to matrices that are square - the number of rows is equal to the number of columns. The alternate method writes the transposed result to a separate destination matrix and can be used on matrices that are not square. The alternate method produces the best performance for very large matrices (the size of the matrix is much larger than the data cache). A thorough discussion of the performance is provided in Section 3.0.

2.0. MATRIX TRANSPOSE FUNCTION

The matrix transpose operation is commonly used to reorder matrix data elements to simplify subsequent calculations. For example, transposing a matrix can simplify a complex calculation on the columns of a large matrix (greater than 32 bytes per row). In order to operate on a matrix column, each element in the column must be read. Since these elements are not adjacent in memory (matrix elements are filled by rows), the processor will read a full cache line (32 bytes) for each column element. It is likely that some of the data will have been replaced in the cache before the operation is performed on subsequent columns, requiring additional reads and incurring additional delays.

However, the calculation can then be performed on the rows of the transpose matrix. This will result in less memory being read into the cache at the start and more efficient use of the entire cache line before the next one is read. The matrix transpose function, illustrated in Figures 1 and 2, is an MMX technology implementation of the matrix transpose operation. It transposes matrices composed of 16-bit elements. The inner loop of the function transposes a 4 x 4 block of data.

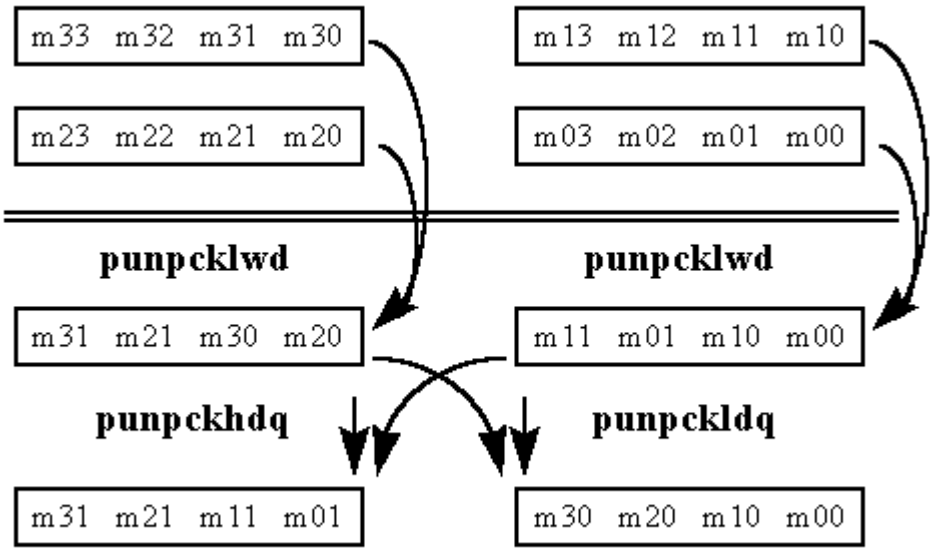

Figure 1. Matrix Transpose in MMX™ Technology - Part I

Figure 1 shows how to transpose the first half of the 4 x 4 block of data. First, the MMX technology PUNPCKLWD instruction is used to interleave the first four elements of rows one and two in the matrix. Next, the PUNPCKLWD instruction is repeated on the first four elements of rows three and four. This produces two intermediate results which are used to generate the first two lines of output. Next the PUNPCKLDQ instruction is used to interleave the intermediate results. This produces the first line to write out (refer to the bottom right of Figure 1). Finally, a PUNPCKHDQ instruction is used on the same intermediate results to produce the second line to write out (refer to the bottom left of Figure 1).

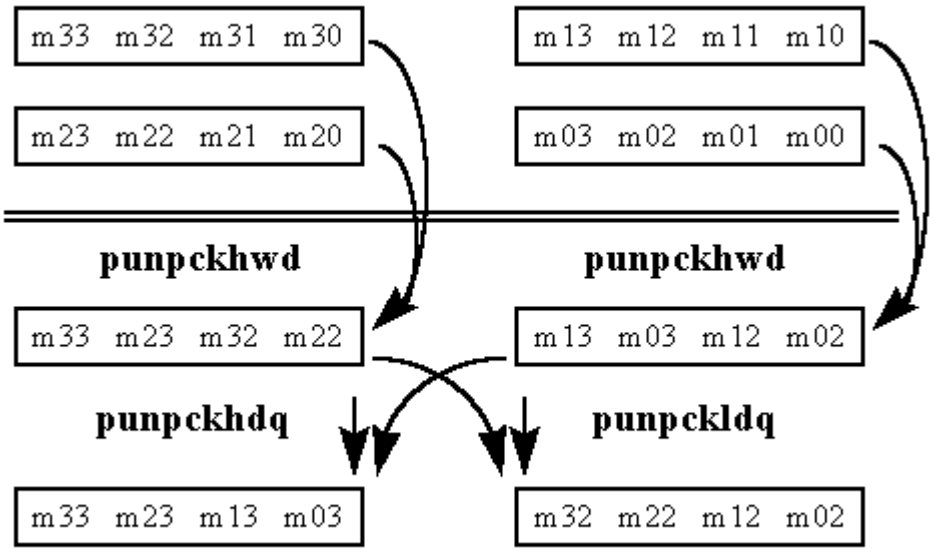

Figure 2. Matrix Transpose Using MMX™ Technology - Part II

Figure 2 shows how to transpose the second half of the 4 x 4 block of data. The first two steps are identical to the process detailed above, except the PUNPCKHWD instruction is substituted for the PUNPCKLWD instruction. This produces two intermediate results which are used to generate the last two lines of output. The PUNPCKLDQ and PUNPCKHDQ instructions are used to produce the final two transposed lines.

2.1. Primary Method

The primary method of implementing the matrix transpose function requires a square matrix, (for non-square matrices see the Alternate Method). The transpose result overwrites the source matrix. This method has the advantage of utilizing the write-back cache feature to defer writing the results to main memory. This allows the transpose operation to execute much faster on matrices which can be contained within the processor's cache.

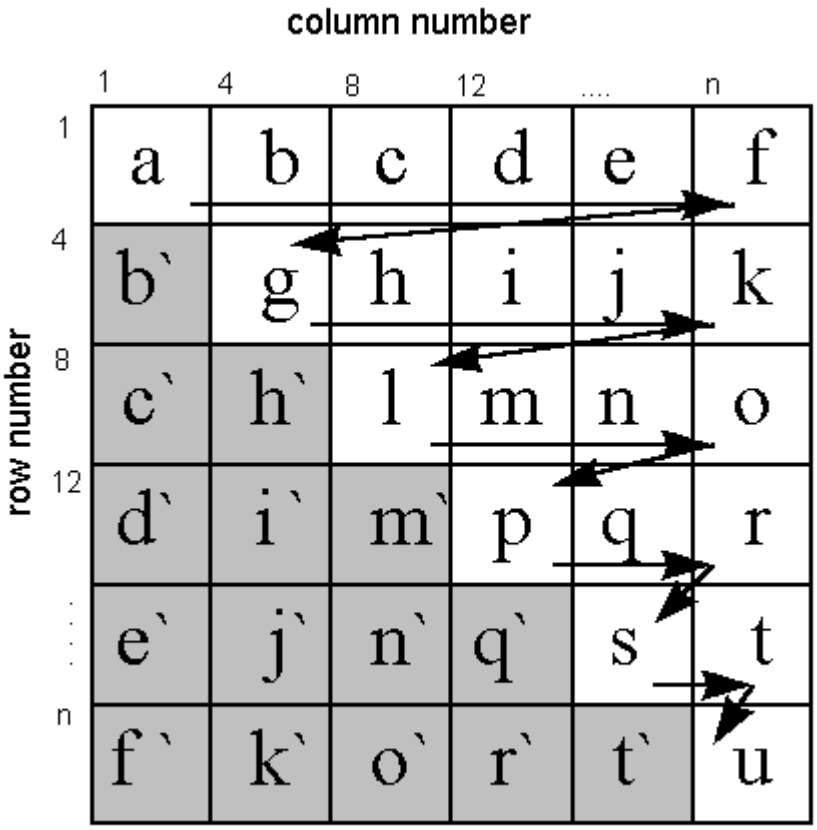

Figure 3. Flow Through the Matrix Transpose Operation - Primary Method

Figure 3 illustrates how a matrix is transposed using the primary method. Each letter in Figure 3 represents a 4 x 4 block of elements in the source matrix. The following is a step by step procedure for transposing the matrix.

1. Set two separate variables to be the number of rows minus four. One of the variables is the outer loop variable, the other is the inner loop variable. Set a pointer to the address of the start of the matrix.

2. For the first block ("a" the first time) follow the algorithm described above for transposing a 4 x 4 block of elements in a matrix. Check to see if the outer loop variable is zero. If it is zero the transpose is complete. Perform the EMMS instruction and return from the function. If the outer loop variable is not zero, proceed to the next step.

3. Set a pointer to the address of the next 4 x 4 block in this row ("b" the first time) and set another pointer to the address of the next 4 x 4 block in this column ("b`" the first time). Then transpose each of the two 4 x 4 blocks ("b" and "b`"). Write the transpose of "b" using the pointer to "b`" and write the transpose of "b`" using the pointer to "b".

4. Set one pointer to the next 4 x 4 block in the row (from "b" to "c" in this case). Set the other pointer to the next 4 x 4 block in the column (from "b`" to "c`" in this case).

5. Subtract four from the inner loop variable and check to see if it is zero. If it is zero then set the pointer to the next block to be transposed ("g" the first time) and go to the next step. If the inner loop variable is not zero then go to Step 3.

6. Subtract four from the outer loop variable and set the inner loop variable to equal the outer loop variable. Then go to Step 2.

Note: The same algorithm is used in standard integer assembly code, but the block being transposed is a 2 x 2 element block of data.

2.2. Alternate Method

If the source matrix is not square, the transposed data cannot overwrite the source matrix. It is necessary to have a destination matrix to write the transposed data to. The number of rows and columns must still be a multiple of four. Every 4 x 4 block of data that is transposed in the source matrix can simply be written to the correct location in the destination matrix. This algorithm is much simpler than the one described above. However, the destination matrix is an uninitialized area of memory and will not reside in the cache.

This algorithm only writes to the destination matrix and therefore every 64-bit write from the transpose must go out of the Pentium® processor's caches to main memory. The performance of this method suffers due to the latency of the four 64-bit writes to memory. On the Pentium® Pro Processor, the write to memory will cause the processor to allocate a cache line for the destination matrix values being written. This has two side-effects. First it will cause the processor to read in the entire cache line in which the write resides. Second, it may also cause an existing cache line to be written back to main memory to make space in the cache for this new line. Both of these effects may cause the performance to suffer during the matrix transpose operation.

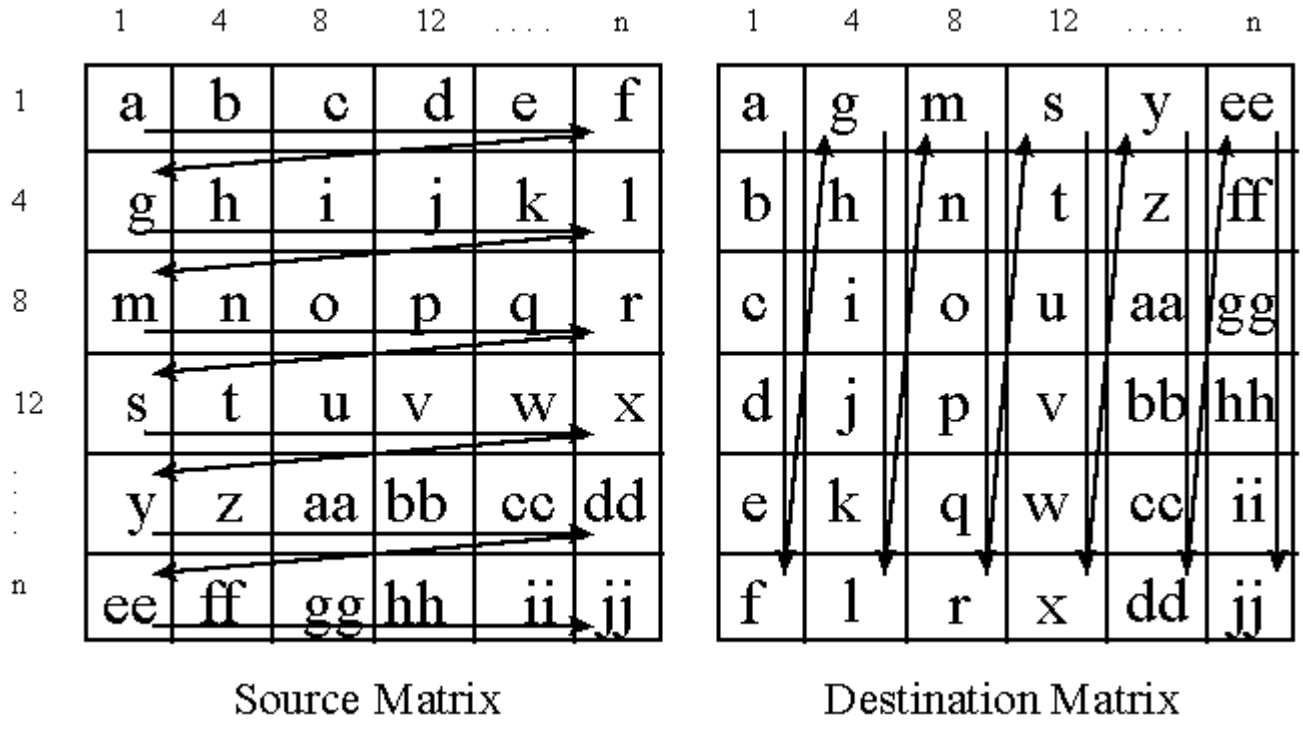

Figure 4. Flow Through the Matrix Transpose Operation - Alternate Method

Figure 4 illustrates how a large matrix is transposed using the alternate method. Each letter in Figure 4 represents a 4 x 4 block of elements in the source and destination matrices. The following is a step by step procedure for transposing the matrix.

1. Set the inner loop variable to the number of columns. Set the outer loop variable to the number of rows. Set a pointer to the address of the start of the source matrix and another pointer to the start of the destination matrix.

2. For the 4 x 4 block of elements pointed to by the source pointer("a" the first time) follow the algorithm described in Section 2.0 for transposing a 4 x 4 block of elements in a matrix. Write out the result to the destination matrix using the destination pointer (block "a" the first time).

3. Set the source pointer to the address of the next 4 x 4 block in this row ("b" the first time) in the source matrix. Set the destination pointer to the address of the next 4 x 4 block in this column ("b" the first time) in the destination matrix.

4. Subtract four from the inner loop variable and check to see if it is zero. If it is zero then set the source pointer to the first block in the next row. Set the destination pointer to the next block in the first row of the destination matrix. Then proceed to the next step. If the inner loop variable is not zero then go to Step 2.

5. Subtract four from the outer loop variable and set the inner loop variable to equal the outer loop variable. Check to see if the outer loop variable is zero. If it is zero then execute EMMS instruction and return. If the outer loop variable is not zero then go to Step 2.

Note: The same algorithm is used in standard integer assembly code, but the block being transposed is a 2 x 2 element block of data.

2.3 Data Alignment

Note that matrix transpose functions do not align the source matrix. The start of the matrix should be aligned to eight bytes beforehand to avoid losing significant performance due to misaligned accesses (an additional three clocks per memory access). Note also that since a 4 x 4 block of data is being transposed, MTMMX can only operate on matrices if their row and column size is a multiple of four words. For applications where this is not the case, two possible solutions are:

- Pad the matrix with columns and rows containing zeroes until their size is a multiple of four before calling MTMMX.
- Alter MTMMX to operate on square matrices of all sizes. One way to do this is to add additional loops before and after the core loop.

3.0. PERFORMANCE GAINS

This section details the performance improvement compared with standard integer assembly code. There is approximately a 2X speedup for the MMX instruction version versus standard assembly code. The results presented here assume all data is in the L1 cache (for small matrices) and aligned to eight bytes - gains are reduced if there are cache misses or misaligned accesses. All performance measurements were taken on a prototype 150MHz Pentium Processor with MMX Technology. The performance numbers are measured in processor clock ticks.

Matrix Size	8 x 8	16 x 16	32 x 32	128 x 128	256 x 256	1024 x 1024
MMX Technology	70 cycles	214 cycles	806 cycles	78 K cycles	590 K cycles	16 M cycles
Scalar Assembly Language	192 cycles	781 cycles	2.9 K cycles	153 K cycles	860 K cycles	32 M cycles

Table 1: Performance Comparison for the Matrix Transpose Operation - Primary Method (Transpose Result Overwrites Source Matrix)

Performance numbers for the primary method are listed in Table 1 for several matrix sizes. The first three columns of data show the performance for matrix sizes which are fully contained in the 16K byte data cache in the processors with MMX Technology. The primary method was found to be the optimal algorithm for transposing a matrix for small square matrices.

The last three columns show the performance for matrix sizes which are much larger than the data cache. The performance of the primary method suffers with very large matrix sizes due to data cache conflict misses. A conflict miss in the data cache occurs when there is no available space in the set (one of four cache locations for a given combination of lower address bits). This causes an existing cache line (32 bytes or 16 matrix elements) to be replaced in the cache by a new line. Since this occurs before all the data in the cache line has been transposed, a single cache line is loaded and written back several times before all of its data is transposed. This results in significant performance degradation for this algorithm.

Matrix Size	8 x 8	16 x 16	32 x 32	128 x 128	256 x 256	1024 x 1024
MMX Technology	215 cycles	572 cycles	2.3 K cycles	49 K cycles	280 K cycles	8.5 M cycles
Scalar Assembly Language	406 cycles	1.7 K cycles	6.6 K cycles	115 K cycles	550 K cycles	17 M cycles

Table 2: Performance Comparison for the Matrix Transpose Operation - Alternate Method (Transpose Result Written to Separate Destination Matrix)

Performance numbers for the alternate method are listed in Table 2 for several matrix sizes. The first three columns of data show the performance for matrix sizes which are fully contained in the 16K byte data cache in the processors with MMX Technology. The performance of the alternate method suffers from write buffer stalls for all sizes of matrices.

A write buffer stall occurs in two situations. The first situation occurs when there is a cache read miss with data in the write buffers, then there is a stall until the write buffers are emptied. The second situation occurs when a write to memory is requested and the write buffers are full (there are four write buffers on the Pentium processor with MMX Technology), then there is a stall until the write buffers are emptied. The alternate method for transposing a matrix produces the second situation for write buffer stalls on the majority of the passes through the inner loop. However, the performance does not degrade significantly as the matrix size increases. This is due to the fact that the entire cache lines are processed for the transpose operation (this prevents evicting and reloading the same cache line). Also the cache line is not modified during the transpose, allowing a new line to simply overwrite an old one without a write back cycle. The consistent performance of the alternate method makes it better suited for transposing very large matrices.

4.0. CODE LISTINGS

The following source code listings are complete assembly language functions which were compiled with Microsoft MASM 6.11d.

4.1 Primary Method

4.1.1 MTMMX Listing

This function transposes a square matrix utilizing MMX Technology instructions. It overwrites the source matrix with the transposed result. All elements in the matrix are 16-bit values.

```
TITLE   mtmmx
;******************************************************************************/
;*
;*      This program has been developed by Intel Corporation.
;*      You have Intel's permission to incorporate this code
;*      into your product, royalty free.  Intel has various
;*      intellectual property rights which it may assert under
;*      certain circumstances, such as if another manufacturer's
;*      processor mis-identifies itself as being "GenuineIntel"
;*      when the CPUID instruction is executed.
;*
;*      Intel specifically disclaims all warranties, express or
;*      implied, and all liability, including consequential and
;*      other indirect damages, for the use of this code,
;*      including liability for infringement of any proprietary
;*      rights, and including the warranties of merchantability
;*      and fitness for a particular purpose.  Intel does not
;*      assume any responsibility for any errors which may
;*      appear in this code nor any responsibility to update it.
;*
;*  *  Other brands and names are the property of their respective
;*     owners.
;*
;*  Copyright (c) 1996, Intel Corporation.  All rights reserved.
;******************************************************************************/
;  This program was assembled with Microsoft MASM 6.11d
;
.nolist
INCLUDE iammx.inc                   ; IAMMX Emulator Macros
.list
.586
.model FLAT
;*****************************************************************************
;    Data Segment Declarations
;*****************************************************************************
.data
;*****************************************************************************
;    Constant Segment Declarations
;*****************************************************************************
.const
;*****************************************************************************
;    Code Segment Declarations
;*****************************************************************************
.code
COMMENT ^
void MTmmx (
```

```
       int16 matr[Y_SIZE][X_SIZE],
       //int16 trans_mmx[X_SIZE][Y_SIZE],
       int , /* X_SIZE */
       int   /* Y_SIZE */ ) ;
^
MTmmx PROC NEAR C USES edi esi ebx eax ecx, ;edx,
            matr:PTR SWORD, ;trans_mmx:PTR SWORD,
            x_size:PTR SWORD, y_size:PTR SWORD
; the 32-bit registers are used only for pointers and loop counters
; all matrix data is held in MMX registers
; eax - holds the (# of rows)-4, used as the inner loop variable
; ebx - hold the number of rows, columns to be transposed,
;       used as an address generation variable
; ecx - set to 6*(# of rows), used to generate the address of
;       the fourth row
; edx - holds the (# of rows)-4, used as the outer loop variable
; esi - set to point to the start of the matrix, used as the
;       secondary pointer in second sub loop
; edi - set to point to the start of the matrix, used as the
;       primary pointer in second sub loop
;
; the MMX registers have no special significance, they are reused
; as soon as they are available
; the comments after MMX instructions show the result in the
; register after the operation is complete in the format:
; m(row number)(col number) for each of the four elements in the register
; the row number and col number are with respect to the 4 x 4
; block of data being transposed
; load the 32-bit registers with their initial values
       mov          ebx, x_size   ; ebx is x_size
       mov          edi, matr     ; pointer to the matrix

       mov          ecx, ebx
       mov          esi, edi      ; pointer to the matrix
       sal          ecx, 2
       mov          eax, ebx
       add          ecx, ebx
       sub          eax, 4        ; eax is the inner loop variable
       add          ecx, ebx      ; ecx is 6*row size
       mov          edx, eax      ; edx is the outer loop variable

do_4x4_block_where_x_equals_y:
       movq   mm0, [esi]          ; m03:m02|m01:m00 - first line
       movq   mm2, [esi+4*ebx]    ; m23:m22|m21:m20 - third line
       movq   mm6, mm0            ; copy first line
       punpcklwd mm0, [esi+2*ebx]
       ; m11:m01|m10:m00 - interleave first and second lines
       movq   mm7, mm2            ; copy third line
       punpcklwd mm2, [esi+ecx]
       ; m31:m21|m30:m20 - interleave third and fourth lines
       movq    mm4, mm0           ; copy first intermediate result
       movq   mm1, [esi+2*ebx]    ; m13:m12|m11:m10 - second line
       punpckldq mm0, mm2
       ; m30:m20|m10:m00 - interleave to produce result 1
       movq   mm3, [esi+ecx]            ; m33:m32|m31:m30 - fourth line
       punpckhdq mm4, mm2
       ; m31:m21|m11:m01 - interleave to produce result 2
       movq   [esi], mm0          ; write result 1
       punpckhwd mm6, mm1
       ; m13:m03|m12:m02 - interleave first and second lines
       movq   [esi+2*ebx], mm4    ; write result 2
       punpckhwd mm7, mm3
       ; m33:m23|m32:m22 - interleave third and fourth lines
       movq   mm5, mm6            ; copy first intermediate result
```

```
        punpckldq mm6, mm7
        ; m32:m22|m12:m02 - interleave to produce result 3
        lea    edi, [edi+8*ebx]
        ; reload edi to point to a 4x4 set 4 rows down
        punpckhdq mm5, mm7
        ; m33:m23|m13:m03 - interleave to produce result 4
        movq   [esi+4*ebx], mm6    ; write result 3
        movq   [esi+ecx], mm5            ; write result 4
        cmp    edx, 0
        ; check to see if the number of rows left is zero
        je     all_done_ready_to_exit
        ;last time through you are done and ready to exit
do_4x4_blocks_x_and_y_not_equal:
; transpose the two mirror image 4x4 sets so that the writes
; can be done without overwriting unused data
        movq   mm0, [esi+8]        ; m03:m02|m01:m00 - first line
        movq   mm2, [esi+4*ebx+8]  ; m23:m22|m21:m20 - third line
        movq   mm6, mm0            ; copy first line
        punpcklwd mm0, [esi+2*ebx+8]
        ; m11:m01|m10:m00 - interleave first and second lines
        movq   mm7, mm2            ; copy third line
        punpcklwd mm2, [esi+ecx+8]
        ; m31:m21|m30:m20 - interleave third and fourth lines
        movq    mm4, mm0           ; copy first intermediate result
; all references for second 4 x 4 block are referred by "n" instead of "m"
        movq   mm1, [edi]          ; n03:n02|n01:n00 - first line
        punpckldq mm0, mm2
        ; m30:m20|m10:m00 - interleave to produce first result
        movq   mm3, [edi+4*ebx]    ; n23:n22|n21:n20 - third line
        punpckhdq mm4, mm2
        ; m31:m21|m11:m01 - interleave to produce second result
        punpckhwd mm6, [esi+2*ebx+8]
        ; m13:m03|m12:m02 - interleave first and second lines
        movq   mm2, mm1            ; copy first line
        punpckhwd mm7, [esi+ecx+8]
        ; m33:m23|m32:m22 - interleave third and fourth lines
        movq   mm5, mm6            ; copy first intermediate result
        movq   [edi], mm0          ; write result 1
        punpckhdq mm5, mm7
        ; m33:m23|m13:m03 - produce third result
        punpcklwd mm1, [edi+2*ebx]
        ; n11:n01|n10:n00 - interleave first and second lines
        movq   mm0, mm3            ; copy third line
        punpckhwd mm2, [edi+2*ebx]
        ; n13:n03|n12:n02 - interleave first and second lines
        movq   [edi+2*ebx], mm4    ; write result 2 out
        punpckldq mm6, mm7
        ; m32:m22|m12:m02 - produce fourth result
        punpcklwd mm3, [edi+ecx]
        ; n31:n21|n30:n20 - interleave third and fourth lines
        movq   mm4, mm1            ; copy first intermediate result
        movq   [edi+4*ebx], mm6    ; write result 3 out
        punpckldq mm1, mm3
        ; n30:n20|n10:n00 - produce first result
        punpckhwd mm0, [edi+ecx]
        ; n33:n23|n32:n22 - interleave third and fourth lines
        movq   mm6, mm2            ; copy second intermediate result
        movq   [edi+ecx], mm5            ; write result 4 out
        punpckhdq mm4, mm3
        ; n31:n21|n11:n01- produce second result
        movq   [esi+8], mm1
        ; write result 5 out - (first result for other 4 x 4 block)
        punpckldq mm2, mm0
        ; n32:n22|n12:n02- produce third result
```

```
        movq   [esi+2*ebx+8], mm4  ; write result 6 out
        punpckhdq mm6, mm0
        ; n33:n23|n13:n03 - produce fourth result
        movq   [esi+4*ebx+8], mm2  ; write result 7 out
        movq   [esi+ecx+8], mm6    ; write result 8 out

        add    esi, 8
        ; increment esi to point to next 4 x 4 block in same row
        lea    edi, [edi+8*ebx]
        ; increment edi to point to next 4 x 4 block below current one
        sub    eax, 4              ; decrement inner loop variable
        jnz     do_4x4_blocks_x_and_y_not_equal
        ; edi points to start of the second row in block we just finished

        sal    edx, 1
        lea    esi, [esi+8*ebx+8]  ; reload edi to point four rows down
        sub    esi, edx
        ; subtract the number of bytes in last row
        ; now we point to spot where row = col
        sub    edx, 8              ; sub 4 from row number
        sar    edx, 1
        mov    edi, esi
        mov    eax, edx
        ; reset x_size to outer loop variable to start new row
        jmp    do_4x4_block_where_x_equals_y
all_done_ready_to_exit:
        emms
        ret
MTmmx ENDP
END
```

4.1.2 MTASM Listing

This function transposes a square matrix utilizing standard integer assembly code instructions. It overwrites the source matrix with the transposed result. All elements in the matrix are 16-bit values.

```
TITLE   mtasm
;*************************************************************************/
;*
;*      This program has been developed by Intel Corporation.
;*      You have Intel's permission to incorporate this code
;*      into your product, royalty free.  Intel has various
;*      intellectual property rights which it may assert under
;*      certain circumstances, such as if another manufacturer's
;*      processor mis-identifies itself as being "GenuineIntel"
;*      when the CPUID instruction is executed.
;*
;*      Intel specifically disclaims all warranties, express or
;*      implied, and all liability, including consequential and
;*      other indirect damages, for the use of this code,
;*      including liability for infringement of any proprietary
;*      rights, and including the warranties of merchantability
;*      and fitness for a particular purpose.  Intel does not
;*      assume any responsibility for any errors which may
;*      appear in this code nor any responsibility to update it.
;*
;*  *  Other brands and names are the property of their respective
;*     owners.
;*
;*  Copyright (c) 1996, Intel Corporation.  All rights reserved.
;*************************************************************************/
```

```
;  This program was assembled with Microsoft MASM 6.11d
;
.list
.586
.model FLAT
;*****************************************************************************
;    Data Segment Declarations
;*****************************************************************************
.data
; inner_loop_var - holds the (# of rows)-2
; outer_loop_var - holds the (# of cols)-2
; temp1 - used to hold the first result in the second sub loop
; temp2 - used to hold the second result in the second sub loop
;
inner_loop_var   dword    ?
outer_loop_var   dword    ?
temp1                     dword    ?
temp2                     dword    ?
;*****************************************************************************
;    Constant Segment Declarations
;*****************************************************************************
.const
;*****************************************************************************
;    Code Segment Declarations
;*****************************************************************************
.code
COMMENT ^
void MTasm (
        int16 matr[Y_SIZE][X_SIZE],
        int , /* X_SIZE */
        int   /* Y_SIZE */ ) ;
^
MTasm PROC NEAR C USES edi esi ebx eax ecx, ;edx,
            matr:PTR SWORD,
            x_size:PTR SWORD, y_size:PTR SWORD
; eax - first line in 2 x 2 block
; ebx - second line in 2 x 2 block
; ecx - address offset for source matrix
;       set to number of columns
; edx - holds a copy of first line in 2 x 2 block
; esi - set to the start of the matrix, used as the
;       primary pointer in the second sub loop
; edi - set to the start of the matrix, used as the
;       secondary pointer in the second sub loop
;
; the comments after instructions show the result in the
; register after the operation is complete in the format:
; m(row number)(col number) for each of the two elements in the register
; the row number and col number are with respect to the 2 x 2
; block of data being transposed
; load the 32-bit registers with their initial values
        mov             ebx, x_size                     ; set ecx to number of columns
        mov             esi, matr
        ; load esi with pointer to source matrix
        sub             ebx, 2
        mov             edi, esi
        ; load edi with pointer to source matrix
        mov             inner_loop_var, ebx
        mov             ecx, x_size                     ; set ecx to number of columns
        mov             edx, ebx
        mov             outer_loop_var, ebx
do_2x2_block_where_x_equals_y:
;first time through this is always true
        mov             ebx, [esi+2*ecx] ; m11:m10 - load second line
```

```
        mov             eax, [esi]              ; m01:m00 - load first line
        shl             ebx, 16                 ; m10:00  - shift low word to top half
        and             eax, 0000ffffh          ; 00:m00  - empty top half
        add             eax, ebx        ;m10:m00 - produce first result
        mov             edx, [esi]              ; m01:m00 - reload first line
        mov             ebx, [esi+2*ecx] ; reload second line
        mov             [esi], eax              ; write result 1 out
        shr             edx, 16                 ; 00:m01  - shift high word to bottom half
        and             ebx, 0ffff0000h         ; m11:00  - empty bottom half
        add             ebx, edx        ; m11:m01 - produce second result
        lea             edi, [edi+4*ecx]
        ; reload edi to point to a 2 x 2 set 2 rows down
        mov             [esi+2*ecx], ebx ; write result 2 out
        add             esi, 4
        ; reload esi to point to a 2 x 2 set in the same row
        cmp             inner_loop_var, 0
        ; check to see if the number of rows left is zero
        je              all_done_ready_to_exit
;last time through you are done and ready to exit
do_2x2_blocks_x_and_y_not_equal:
; transpose the two mirror image 2x2 sets so that the writes
; can be done without overwriting unused data
        mov             ebx, [esi+2*ecx] ; m11:m10 - load second line
        mov             eax, [esi]              ; m01:m00 - load first line
        shl             ebx, 16                 ; m10:00  - shift low word to top half
        and             eax, 0000ffffh          ; 00:m00  - empty top half
        add             eax, ebx        ; m10:m00 - produce first result
        mov             ebx, [esi+2*ecx] ; reload second line
        mov             temp1, eax              ; write result 1 to a temporary spot
        mov             eax, [esi]              ; reload first line
        shr             eax, 16                 ; 00:m01  - shift high word to bottom half
        and             ebx, 0ffff0000h         ; m11:00  - empty bottom half
; all references for second 2 x 2 block are referred by "n" instead of "m"
        add             ebx, eax        ; m11:m01 - produce second result
        mov             eax, [edi]              ; n01:n00 - load first line
        mov             temp2, ebx              ; write result 2 to a temporary spot
        mov             ebx, [edi+2*ecx] ; n11:n10 - load second line
        shl             ebx, 16                 ; n10:00  - shift low word to top half
        and             eax, 0000ffffh          ; 00:n00  - empty top half
        add             eax, ebx        ; n10:n00 - produce third result
        mov             ebx, [edi+2*ecx] ; reload second line
        mov             [esi], eax              ; write result 3 out
        mov             eax, [edi]              ; reload first line
        shr             eax, 16                 ; 00:n01  - shift high word to bottom half
        and             ebx, 0ffff0000h         ; n11:00  - empty bottom half
        add             ebx, eax        ; n11:n01 - produce fourth result
        mov             eax, [edi]
        mov             [esi+2*ecx], ebx ; write result 4 out
        mov             eax, temp1
        mov             ebx, temp2
        mov             [edi], eax              ; write result 1 out
        mov             [edi+2*ecx], ebx ; write result 2 out
        lea             edi, [edi+4*ecx]
        ; increment edi to point to next 2 x 2 block below current one
        add     esi, 4
        ; increment esi to point to next 2 x 2 block in same row
        sub     inner_loop_var, 2       ; decrement inner loop variable
        jnz     do_2x2_blocks_x_and_y_not_equal
        ; edi points to start of the second row in block we just finsished
        mov             edx, outer_loop_var
        shl             edx, 1
        lea             esi, [esi+4*ecx] ; reload edi to point four rows down
        sub             esi, edx
        ; subtract the number of bytes in last row
```

```
        ; now we point to spot where row = col
        sub             edx, 4                  ; sub 4 from number of cols
        shr             edx, 1
        mov             edi, esi
        mov             outer_loop_var, edx
        mov             inner_loop_var, edx
        ; reset inner_loop_var to outer loop variable to start new row
        jmp             do_2x2_block_where_x_equals_y
all_done_ready_to_exit:
        ret
MTasm ENDP
END
```

4.2 Alternate Method

4.2.1 MTMMX Listing

This function transposes a matrix utilizing MMX Technology instructions. It writes the transposed result to a different matrix. All elements in the matrix are 16-bit values.

```
TITLE   mtmmx
;*************************************************************************/
;*
;*      This program has been developed by Intel Corporation.
;*      You have Intel's permission to incorporate this code
;*      into your product, royalty free.  Intel has various
;*      intellectual property rights which it may assert under
;*      certain circumstances, such as if another manufacturer's
;*      processor mis-identifies itself as being "GenuineIntel"
;*      when the CPUID instruction is executed.
;*
;*      Intel specifically disclaims all warranties, express or
;*      implied, and all liability, including consequential and
;*      other indirect damages, for the use of this code,
;*      including liability for infringement of any proprietary
;*      rights, and including the warranties of merchantability
;*      and fitness for a particular purpose.  Intel does not
;*      assume any responsibility for any errors which may
;*      appear in this code nor any responsibility to update it.
;*
;*  *  Other brands and names are the property of their respective
;*     owners.
;*
;*  Copyright (c) 1996, Intel Corporation.  All rights reserved.
;*************************************************************************/
;  This program was assembled with Microsoft MASM 6.11d
;
.nolist
INCLUDE iammx.inc                   ; IAMMX Emulator Macros
.list
.586
.model FLAT
;*****************************************************************
;    Data Segment Declarations
;*****************************************************************
.data
four_times_columns DWORD ?                  ; set to (# of columns)*4
six_times_columns DWORD ?           ; set to (# of columns)*6
;*****************************************************************
;    Constant Segment Declarations
;*****************************************************************
.const
```

```
;*****************************************************************
;    Code Segment Declarations
;*****************************************************************
.code
COMMENT ^
void MTmmx (
        int16 matr[Y_SIZE][X_SIZE],
        int16 trans_mmx[X_SIZE][Y_SIZE],
        int , /* X_SIZE */
        int   /* Y_SIZE */ ) ;
^
MTmmx PROC NEAR C USES  edi esi ebx eax ecx edx,
            matr:PTR SWORD, trans_mmx:PTR SWORD,
            x_size:PTR SWORD, y_size:PTR SWORD
; the 32-bit registers are used only for pointers and loop counters
; all matrix data is held in MMX registers
; eax - loop variable initially set to number of columns
;       decremented by 4 each time through the inner loop
; ebx - address offset for destination matrix
;       set to number of rows
; ecx - address offset for source matrix
;       set to number of columns
; edx - loop variable initially set to number of rows
;       decremented by 4 each time through the outer loop
; esi - pointer to source matrix
; edi - pointer to destination matrix
;
; mm0 - first line in 4 x 4 block
; mm1 - second line in 4 x 4 block
; mm2 - third line in 4 x 4 block
; mm3 - fourth line in 4 x 4 block
; mm4 - copy of first line
; mm5 - copy of third line
; mm6 - copy of first intermediate result
; mm7 - copy of third intermediate result
;
; the comments after MMX instructions show the result in the
; register after the operation is complete in the format:
; m(row number)(col number) for each of the four elements in the register
; the row number and col number are with respect to the 4 x 4
; block of data being transposed
; load the 32-bit registers with their initial values
        mov     ecx, x_size             ; set ecx to number of columns
        mov     ebx, y_size             ; set ebx to number of rows
        mov     eax, ecx
        ; set eax to number of columns, inner loop variable
        lea     edx, [ecx+2*ecx]
        mov     esi, matr
        ; load esi with pointer to source matrix
        add     edx, edx        ; set edx to 6 * number of columns
        mov     edi, trans_mmx
        ; load edi with pointer to destination matrix
        shl     ecx, 2
        mov     six_times_columns, edx
        mov     four_times_columns, ecx
        mov     edx, ebx
        ; set edx to number of rows, outer loop variable
        shr     ecx, 2
do_next_row_of_matr:
do_next_4x4_block:
        ; load the 4x4 block into mmx regs

        movq    mm0, [esi]              ; m03:m02|m01:m00 - load first line
```

```
        movq    mm1, [esi+2*ecx] ; m13:m12|m11:m10 - load second line
        movq    mm4, mm0         ; copy first line
        lea     esi, [esi+4*ecx] ; set esi to point to third line
        punpcklwd mm0, mm1
        ; m11:m01|m10:m00 - interleave first and second lines
        movq    mm2, [esi]              ; m23:m22|m21:m20 - load third line
        punpckhwd mm4, mm1
        ; m13:m03|m12:m02 - interleave first and second lines
        movq    mm5, mm2         ; copy third line
        punpcklwd mm2, [esi+2*ecx]
        ; m31:m21|m30:m20 - interleave third and fourth lines
        movq    mm6, mm0         ; copy first intermediate result
        punpckldq mm0, mm2
        ; m30:m20|m10:m00 - interleave to produce result 1
        punpckhwd mm5, [esi+2*ecx]
        ; m33:m23|m32:m22 - interleave third and fourth lines
        movq    mm7, mm4         ; copy third intermediate result
        punpckhdq mm6, mm2
        ; m31:m21|m11:m01 - interleave to produce result 2
        sub     esi, four_times_columns  ; reset esi to original point
        movq    [edi], mm0               ; write result 1 out
        punpckldq mm4, mm5
        ; m32:m22|m12:m02 - interleave to produce result 3
        movq    [edi+2*ebx], mm6 ; write result 2 out
        punpckhdq mm7, mm5
        ; m32:m22|m12:m02 - interleave to produce result 4
        lea     edi, [edi+4*ebx]
        ; set edi to point to the third line in 4 x 4 block
        add     esi, 8
        ; esi to point to next 4 x 4 block in source
        sub     eax, 4                   ; sub 4 from inner loop variable

        movq    [edi], mm4               ; write result 3 out
        movq    [edi+2*ebx], mm7 ; write result 4 out
        lea     edi, [edi+4*ebx]
        ; set edi to point to next 4 x 4 block in destination
        jnz     do_next_4x4_block
        mov     edi, trans_mmx
        ; reset edi to point to start of destination
        add     esi, six_times_columns
        ; set esi to point to next row of 4 x 4 blocks
        add     edi, 8
        ; edi points to next 4 x 4 block in destination
        sub     edx, 4                   ; sub 4 from outer loop variable
        mov     trans_mmx, edi
        ; change the start of trans_mmx to be this column
        mov     eax, ecx         ; reset inner loop variable
        jnz     do_next_row_of_matr
    emms
    ret
MTmmx ENDP
END
```

4.2.2 MTASM Listing

This function transposes a matrix utilizing standard integer assembly language instructions. It writes the transposed result to a different matrix. All elements in the matrix are 16-bit values.

```
TITLE   mtasm
;*************************************************************************/
;*
;*      This program has been developed by Intel Corporation.
;*      You have Intel's permission to incorporate this code
```

```
;*      into your product, royalty free.  Intel has various
;*      intellectual property rights which it may assert under
;*      certain circumstances, such as if another manufacturer's
;*      processor mis-identifies itself as being "GenuineIntel"
;*      when the CPUID instruction is executed.
;*
;*      Intel specifically disclaims all warranties, express or
;*      implied, and all liability, including consequential and
;*      other indirect damages, for the use of this code,
;*      including liability for infringement of any proprietary
;*      rights, and including the warranties of merchantability
;*      and fitness for a particular purpose.  Intel does not
;*      assume any responsibility for any errors which may
;*      appear in this code nor any responsibility to update it.
;*
;*  *  Other brands and names are the property of their respective
;*     owners.
;*
;*  Copyright (c) 1996, Intel Corporation.  All rights reserved.
;*****************************************************************************/
;  This program was assembled with Microsoft MASM 6.11d
;
.586
.model FLAT
;*****************************************************************************
;    Data Segment Declarations
;*****************************************************************************
.data
inner_loop_var DWORD ?
outer_loop_var DWORD ?
;*****************************************************************************
;    Constant Segment Declarations
;*****************************************************************************
.const
;*****************************************************************************
;    Code Segment Declarations
;*****************************************************************************
.code
COMMENT ^
void MTasm (
       int16 matr[Y_SIZE][X_SIZE],
       int16 trans_asm[X_SIZE][Y_SIZE],
       int , /* X_SIZE */
       int   /* Y_SIZE */ ) ;
^
MTasm PROC NEAR C USES  edi esi ebx eax ecx edx,
            matr:PTR SWORD, trans_asm:PTR SWORD,
            x_size:PTR SWORD, y_size:PTR SWORD
; eax - first line in 2 x 2 block
; ebx - second line in 2 x 2 block
; ecx - address offset for source matrix
;       set to number of columns
; edx - address offset for destination matrix
;       set to number of rows
; esi - pointer to source matrix
; edi - pointer to destination matrix
;
; the comments after instructions show the result in the
; register after the operation is complete in the format:
; m(row number)(col number) for each of the two elements in the register
; the row number and col number are with respect to the 2 x 2
; block of data being transposed
; load the 32-bit registers with their initial values
       mov    ecx, x_size         ; set ecx to number of columns
```

```
      mov    edx, y_size         ; set ebx to number of rows
      mov    esi, matr           ; load esi with pointer to source matrix
      mov    edi, trans_asm            ; load edi with pointer to destination matrix
      mov    inner_loop_var, ecx ; set to the number of columns
      mov    outer_loop_var, edx ; set to the number of rows
do_next_row_of_matr:
do_next_2x2_block:
      ; load the 2 x 2 block into regs
      mov    ebx, [esi+2*ecx]    ; m11:m10 - load second line
      mov    eax, [esi]          ; m01:m00 - load first line
      shl    ebx, 16                   ; m10:00  - shift low word up to top half
      and    eax, 0000ffffh            ; 00:m00  - empty top half
      add    eax, ebx            ; m10:m00 - produce first result
      mov    ebx, [esi+2*ecx]    ; reload second line
      mov    [edi], eax          ; write result 1 out
      mov    eax, [esi]          ; reload first line
      shr    eax, 16                   ; 00:m01  - shift high word down to bottom half
      and    ebx, 0ffff0000h           ; m11:00  - empty bottom half
      add    ebx, eax            ; m11:m01 - produce second result
      mov    [edi+2*edx], ebx    ; write result 2 out
      lea    edi, [edi+4*edx]
      ; set edi to point to the first line in next 2 x 2 block
      add     esi, 4
      ; set esi to point to next 2 x 2 block in source
      sub     inner_loop_var, 2  ; sub 4 from inner loop variable
      jnz     do_next_2x2_block
      mov    edi, trans_asm
      ; reset edi to point to start of destination
      lea    esi, [esi+2*ecx]
      ; set esi to point to next row of 2 x 2 blocks
      add    edi, 4
      ; edi points to next 2 x 2 block in destination
      sub    outer_loop_var, 2   ; sub 4 from y_size
      mov    trans_asm, edi
      ; change the start of trans_asm to be this column
      mov    inner_loop_var, ecx ; reset inner loop variable
      jnz    do_next_row_of_matr
   ret
MTasm ENDP
END
```